CS1101: Foundations of Programming

File handling

Dept. of Computer Science & Engineering

Indian Institute of Technology Patna

What is a file?

- **A named collection of data, stored in secondary storage usually**
- **Typical operations on files:**
  - **Open**
  - **Read**
  - **Write**
  - **Close**
- **How is a file stored?**
  - **Stored as sequence of bytes, logically contiguous, need not be physically contiguous on disk**
  - **C gives us a simplified view of a file stored on the disk**

File types

- **Primarily two kinds of files**
  - **Text — Human readable files**
    - **A text editor can show the contents of a text file**
  - **Binary — Primarily not meant for human reading**
    - **Executable, video, audio, image files**
    - **You need special programs to view / process the file meaningfully (like, need some image viewer tool to see images, similarly video player, etc.)**

File handling in C

- In C, we use $FILE*$ to represent a pointer to file
- `fopen()` is used to open a file. It returns the special value NULL to indicate that it is unable to open the file

```
FILE *fptr;
char filename[] = "myfile.dat";
fptr = fopen(filename, "w");
if(fptr == NULL){
   printf("Error in file creation\n");
   /* do something as per need */
}
```

Modes for opening files

- **The second argument in** `fopen` **is the** **mode** **in which we open the file. There are three basic modes**
  - **"`r`" opens a file for reading**
  - **"`r+`" allows write**
  - **"`w`" creates a file for writing and writes over all previous contents, deletes any previous file of the same name (CAREFUL)**
  - **"`w+`" allows read**
  - **"`a`" opens file for appending – writing at the end of file, previous content remain intact**
  - **"`a+`" allows read**

The `exit()` **function**

- **Sometime error checking means we want an immediate exit from a program**
- **In `main()` function we can use return to stop the execution of the program**
- **In any function we can use `exit()` to do this**
- `exit()` **is declared in** `stdlib.h`
  - `exit(1);` **// to indicate status of exit**
  - `exit(2);`
  - `exit(3);`
  - **…**

Usage of `exit()` function

```c
FILE *fptr;
char filename[]="myfile.dat";
fptr = fopen(filename, "w");
if(fptr == NULL){
    printf("Error in file creation\n");
    exit(0);
}
```

Writing to a file using `fprintf()`

- `fprintf()` **works just like the** `printf()` **except that its first argument is a file pointer**

```c
FILE *fptr;
int a=3, b=8;
char filename[]="myfile.dat";
fptr = fopen(filename, "w");
fprintf(fptr, "First file\n");
fprintf(fptr, "%d %d\n", a, b);
```

Reading data from a file using `fscanf()`

- **The C library maintains a file-pointer to remember the position up to which a file has been read so far. The file-pointer moves forward with each read operation**

- **Next read operation (e.g., `fscanf`, `fgets`, `fgetc`) will give the contents of the file after this position.**

- **Each function for reading from a file has a way to inform that the end of file has been reached (usually by returning a special value like NULL or EOF)**

```
FILE *fptr;
int a, b;
fptr = fopen("myfile.dat", "r");
fscanf(fptr, "%d%d", &a, &b);
```

EOF (End of file)

- `EOF` **is a special value that signifies that the file pointer has reached the end of the file stream.**

- `EOF` **is returned by** `fgetc()` **and** `fscanf()` **if the end of file has been reached.**

- **Another way to detect the end of a file is to use the call** `feof(fp)`.

- `feof()` **returns** `true` **only after a read operation from the file fails.**

- `feof()` **cannot probe and notify that the next read operation will fail. This is oftentimes not possible because whether the end-of-file is reached or not depends on what you plan to read next (like an** `int` **or a** `char`**).**

Example of EOF (End of file)

- **Let us assume the task is to read a file containing a set of integers and print the average of those numbers**

- **Consider that the last number is 10 in file** `input.txt` **and then it has few more new lines and spaces**

- **So,** `%d` **reading will fail but a** `%c` **will not**

- **You need to attempt a reading and then check for** `EOF` **of** `feof()`

input.txt:
43 24 9 0
23 45 77

34 12
10

Checking of end-of-file using EOF

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    FILE *fp; int n, sum, x;
    fp = (FILE *)fopen("input.txt", "r");
    if(fp == NULL) exit(1);
    n = sum = 0;
    while (1) {
        if (fscanf(fp, "%d", &x) == EOF) break;
        ++n; sum += x;
        printf("%d-th integer: %d\n", n, x);
    }
    fclose(fp);
    printf("Average: %.2f\n", ((float)sum)/n);
    exit(0);
}
```

Output:
```
1-th integer: 43
2-th integer: 24
3-th integer: 9
4-th integer: 0
5-th integer: 23
6-th integer: 45
7-th integer: 77
8-th integer: 34
9-th integer: 12
10-th integer: 10
Average: 27.70
```

Checking of end-of-file using EOF

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    FILE *fp; int n, sum, x;
    fp = (FILE *)fopen("input.txt", "r");
    if(fp == NULL) exit(1);
    n = sum = 0;
    while (1) {
        fscanf(fp, "%d", &x); if(feof(fp)) break;
        ++n; sum += x;
        printf("%d-th integer: %d\n", n, x);
    }
    fclose(fp);
    printf("Average: %.2f\n", ((float)sum)/n);
    exit(0);
}
```

Output:
```
1-th integer: 43
2-th integer: 24
3-th integer: 9
4-th integer: 0
5-th integer: 23
6-th integer: 45
7-th integer: 77
8-th integer: 34
9-th integer: 12
10-th integer: 10
Average: 27.70
```

Checking of end-of-file using EOF: Wrong program

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
    FILE *fp; int n, sum, x;
    fp = (FILE *)fopen("input.txt", "r");
    if(fp == NULL) exit(1);
    n = sum = 0;
    while (!feof(fp)) {
        fscanf(fp, "%d", &x);
        ++n; sum += x;
        printf("%d-th integer: %d\n", n, x);
    }
    fclose(fp);
    printf("Average: %.2f\n", ((float)sum)/n);
    exit(0);
}
```

Output:
```
1-th integer: 43
2-th integer: 24
3-th integer: 9
4-th integer: 0
5-th integer: 23
6-th integer: 45
7-th integer: 77
8-th integer: 34
9-th integer: 12
10-th integer: 10
11-th integer: 10
Average: 26.09
```

Reading data from file using `fgets()`

- **We can read a string from a file using** `fgets()`
- `fgets()` **takes 3 arguments – a string, maximum number of characters to store in the string, and a file pointer. It returns NULL if there is an error (or end of file is reached)**

```c
FILE *fp;
char str[1000];
fp = (FILE *)fopen("input.txt", "r");
...
while (fgets(str, 1000, fp) != NULL) {
   /* read 999 chars at most at a time */
   printf("String read is: %s\n",str);
}
...
```

Reading data from file using `fgets()`

- **A maximum of size 1 byte will be read from the input file stream**
- **The reading includes the new line character if it appears in these many bytes.**
- `fgets()` **null-terminates the string by putting the null character** `'\0'`.
- **With appropriate size,** `fgets()` **never leads to buffer overflow.**
- **In the example:**
  - **If the line contains at most 998 characters, the entire line and the new-line character will be read and stored in the string.**
  - **If the line contains 999 or more characters, only the first 999 characters will be read and stored in the string.**

Closing a file

- **We can close a file simply by calling** `fclose()` **with the file pointer**

```c
FILE *fp;
fp = (FILE *)fopen("new.txt", "w");
if(fp == NULL){
   printf("Error in file creation\n");
   exit(0);
}
fprintf(fp, "Write whatever you want!\n");
...
fclose(fp);
```

Three special file streams

- **Three special file streams are defined in the header file** `<stdio.h>`**. These** `FILE` **pointers are automatically opened in every program**
  - `stdin` **reads input from the keyboard**
  - `stdout` **send output to the screen**
  - `stderr` **prints errors to an error device (usually also the screen)**

- `scanf(...)` **is the same as** `fscanf(stdin, ...)`

- `printf(...)` **is the same as** `fprintf(stdout, ...)`

Example

```c
#include<stdio.h>
int main(){
   int x;

   fprintf(stdout,"Enter x\n");
   fscanf(stdin,"%d",&x);
   fprintf(stdout,"x: %d\n",x);
   fprintf(stderr,"No-error\n");
   return 0;
}
```

Output:
```
Enter x
43
x: 43
No-error
```

Reading and writing a character

- **A character reading/writing is equivalent to reading/writing a byte**
  - `int getchar();`
  - `int putchar(int c);`
  - `int fgetc(FILE *fp);`
  - `int fputc(int c, FILE *);`

```
char c;
c = getchar();
putchar(c);
```

```
char c;
FILE *fp;
c = fgetc(fp);
fputc(c, fp);
```

Random access using `fseek()`

- `ftell()` — **returns the present position of the file pointer**
  `long ftell(FILE *fp)`
- `fseek()` — **can be used to set the position of a file pointer**
  `int fseek(FILE *fp, long offset, int from_where)`
  - **The new position, measured in bytes, is obtained by adding** `offset` **bytes to the position specified by** `from_where`
  - `from_where` **can take one the following values**
    - `SEEK_END` — **end of the file**
    - `SEEK_SET` — **beginning of the file**
    - `SEEK_CUR` — **current position of the file pointer**

Example

```c
int main(){
    char c; FILE *fp;
    fp = fopen("input.txt", "r+");
    printf("%ld \n", ftell(fp));
    c = fgetc(fp); c = fgetc(fp);
    printf("%ld \n", ftell(fp));
    fseek(fp, 2, SEEK_CUR);
    printf("%ld \n", ftell(fp));
    fputs("fast purple",fp);
    printf("%ld \n", ftell(fp));
    fclose(fp);
    return 0;
}
```

Output:
0
2
4
15

input.txt:

Before: the quick brown fox jumped over the lazy dogs

After: the fast purple fox jumped over the lazy dogs

Example: `fseek()`

- **Assume that we have file having 1000 prime numbers, one number in each line and there are no extra spaces**
- **We open the file in the read mode.**
- **We do not read the file from beginning to end.**
- **When the user specifies some n in the range [1,1000], we go to the location where the n-th prime is stored, and read that prime.**
- **2 is the first prime (not the zero-th prime).**
- **We need to know exactly where the n-th prime is stored.**
- **The following statistics help us do that.**
  - **There are exactly four 1-digit primes.**
  - **There are exactly 21 2-digit primes.**
  - **There are exactly 143 3-digit primes.**
  - **The 1000-th prime is a 4-digit prime.**
- **We must not forget the new line character at the end of each line.**

primes.txt:
2
3
5
7
11
. . .
107
. . .
7919

Example: `fseek()`

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  FILE *fp; int n, p;
  fp = (FILE *)fopen("primes.txt", "r");
  while (1) {
    printf("Which prime? "); scanf("%d", &n);
    if((n < 1) || (n > 1000)) break;
    if(n <= 4) fseek(fp, (n - 1) * 2, SEEK_SET);
    else if(n <= 25) fseek(fp, 4 * 2 + (n - 5) * 3, SEEK_SET);
    else if(n <= 168) fseek(fp, 4 * 2 + 21 * 3 + (n - 26) * 4, SEEK_SET);
    else fseek(fp, 4 * 2 + 21 * 3 + 143 * 4 + (n - 169) * 5, SEEK_SET);
    fscanf(fp, "%d", &p); printf("%d-th prime is %d\n", n, p);
  }
  fclose(fp); exit(0);
}
```

Example: Copying a file

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main() {
  FILE *ifp, *ofp; int i, c; char srcfile[100], dstfile[100];
  strcpy(srcfile, "source.txt"); strcpy(dstfile, "copy.txt");
  if((ifp = (FILE *)fopen(srcfile, "r"))== NULL){
    printf("File does not exist\n"); exit(0);
  }
  if((ofp = (FILE *)fopen(dstfile, "w"))== NULL){
    printf("Unable to create file\n"); exit(0);
  }
  while ( (c = fgetc(ifp)) != EOF) fputc(c, ofp);
  fclose(ifp); fclose(ofp);
  return 0;
}
```

Example: reading & writing binary files

- **We want to store Fibonacci numbers F(0), F(1), ..., F(40) in a file**

- **Text mode**
    - **We store 0, 1, 1, 2, 3, . . . , 102334155, one in a single line.**
    - **Some separators are needed between to consecutive Fibonacci numbers (here we use `'\n'`).**
    - `fseek()` **to locate F(i) will be difficult.**

- **Binary mode**
    - **Assume that `int` is of size 32 bits (4 bytes).**
    - **We store the raw 32 bits of each F(i) one after another.**
    - **No separators are needed because each F(i) occupies exactly 4 bytes.**
    - `fseek()` **to locate F(i) will be easy: just go to the 4i-th byte from the beginning.**

Storing in human-readable format

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
   FILE *fp;
   int n = 40, i, F[41];
   F[0] = 0; F[1] = 1;
   for(i=2; i<=n; ++i) F[i] = F[i-1] + F[i-2];
   fp = (FILE *)fopen("Fib.txt", "w");
   if(fp == NULL) exit(1);
   for(i=0; i<=n; ++i) fprintf(fp, "%d\n", F[i]);
   fclose(fp);
   exit(0);
}
```

Storing in binary format

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  FILE *fp; char *p;
  int n = 40, i, j, F[41];
  F[0] = 0; F[1] = 1;
  for(i=2; i<=n; ++i) F[i] = F[i-1] + F[i-2];
  fp = (FILE *)fopen("FibB.dat", "w");
  if(fp == NULL) exit(1);
  for(i=0; i<=n; ++i) {
    p = (char *)(F + i);
    for(j=0; j<4; ++j) { fprintf(fp, "%c", *p); ++p; }
  }
  fclose(fp);
  exit(0);
}
```

Reading in binary format

```c
#include <stdio.h>
#include <stdlib.h>
int main() {
  FILE *fp; char *p;
  int n = 40, i, j, F[41];
  fp = (FILE *)fopen("FibB.dat", "r");
  for(i=0; i<=n; ++i) {
    p = (char *)(F+i);
    for(j=0; j<4; ++j) { fscanf(fp, "%c", p); ++p; }
  }
  fclose(fp);
  for(i=0; i<=n; ++i) printf("F(%d) = %d\n", i, F[i]);
  exit(0);
}
```

FibH.dat: 220 bytes
FibB.dat: 164 bytes

Redirection: Input & Output file

- **One may redirect the standard input and output to other files, other than stdin or stdout**

- **Suppose, we have an executable code a.out**

  `$ ./a.out < in.txt > out.txt`
  - `scanf()` **will read data from the file** "`in.txt`" **and**
  - `printf()` **will print the results on the file** "`out.txt`"

- **Another option**

  `$ ./a.out < in.txt >> out.txt`
  - `scanf()` **will read data from the file** "`in.txt`" **and**
  - `printf()` **will append the results at the end of file** "`out.txt`"

Command line arguments

- **A program can be executed by directly typing the command at the shell prompt**
  ```
  $> gcc filename.c
  $> ./a.out in.txt out.txt
  $> exe_name param1 param2 ...
  ```
- **The individual items specified are separated from one another by spaces. Use quotes to enter arguments with spaces.**

  - **First item is the executable name**
- **Variables** `argc` **and** `argv` **keep track of the items specified in the command line**
- **Command line arguments may be passed by specifying them under** `main()`
  ```
  int main(int argc, char *argv[]);
  ```
  `argc` **- argument count,** `argv` **- array of string as command line arguments,** `argv[]` **is NULL-terminated**

Example

- `$> ./a.out in.txt out.txt`
  - `argc = 3`
  - `argv[0]="./a.out", argv[1]="in.txt", argv[2]="out.txt", argv[3]=NULL`

Copying file: `./a.out <src> <dst>`

```c
int main(int argc, char *argv[]) {
    FILE *ifp, *ofp; int i, c; char src[100], dst[100];
    if(argc!=3){ printf("Usage: ./a.out <src> <dst>\n"); exit(0); }
    else{ strcpy(src, argv[1]); strcpy(dst, argv[2]); }
    if((ifp = (FILE *)fopen(src, "r"))== NULL){
        printf("File does not exist\n"); exit(0);
    }
    if((ofp = (FILE *)fopen(dst, "w"))== NULL){
        printf("Unable to create file\n"); exit(0);
    }
    while ( (c = fgetc(ifp)) != EOF) fputc(c, ofp);
    fclose(ifp); fclose(ofp);
    return 0;
}
```

Useful library functions

- `$> ./a.out xyz 123 4.56 "fname lname"`
- `strcpy(str, argv[1]);` — **copies string**
- `val = atoi(argv[2]);` — **converts string to integer**
- `cpi = atof(argv[3]);` — **converts string to double**
- `strcpy(name, argv[4]);` — **copies string**

Practice problems

- Write a program that reads a file, converts all lower-case letters to the upper case, and keeps the other characters intact, and stores the output in another file.
- Write a program that reads a 2-d array of integers from a file and replaces the contents of the file with the transpose of the matrix represented by the 2-d array.
- Write a program that reads student records containing name (string), roll_number (int), CGPA (float) from the user and writes them in a file, one record per line.
- Write a program that reads the records written by the above program into an array of structures. The structure should contain name, roll_number and CGPA as members.